Web Server and ASP.NET Application Life Cycle in Depth

Massimiliano Peluso "WeDev Limited", 21 Dec 2010 CPOL

4.87 (115 votes)

This article describes in depth everything that happens beween the request/response in a web application.

Introduction

In this article, we will try to understand what happens when a user submits a request to an ASP.NET web app. There are lots of articles that explain this topic, but none that shows in a clear way what really happens in depth during the request. After reading this article, you will be able to understand:

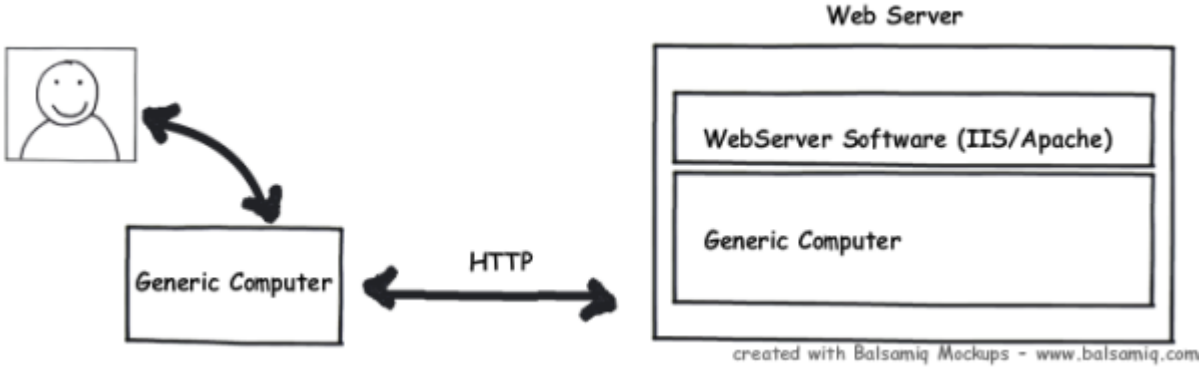
- What is a Web Server
- HTTP - TCP/IP protocol
- IIS
- Web communication
- Application Manager
- Hosting environment
- Application Domain
- Application Pool
- How many app domains are created against a client request
- How many HttpApplications are created against a request and how I can affect this behaviour
- What is the work processor and how many of it are running against a request
- What happens in depth between a request and a response

Start from scratch

All the articles I have read usually begins with "The user sends a request to IIS... bla bla bla". Everyone knows that IIS is a web server where we host our web applications (and much more), but what is a web server?

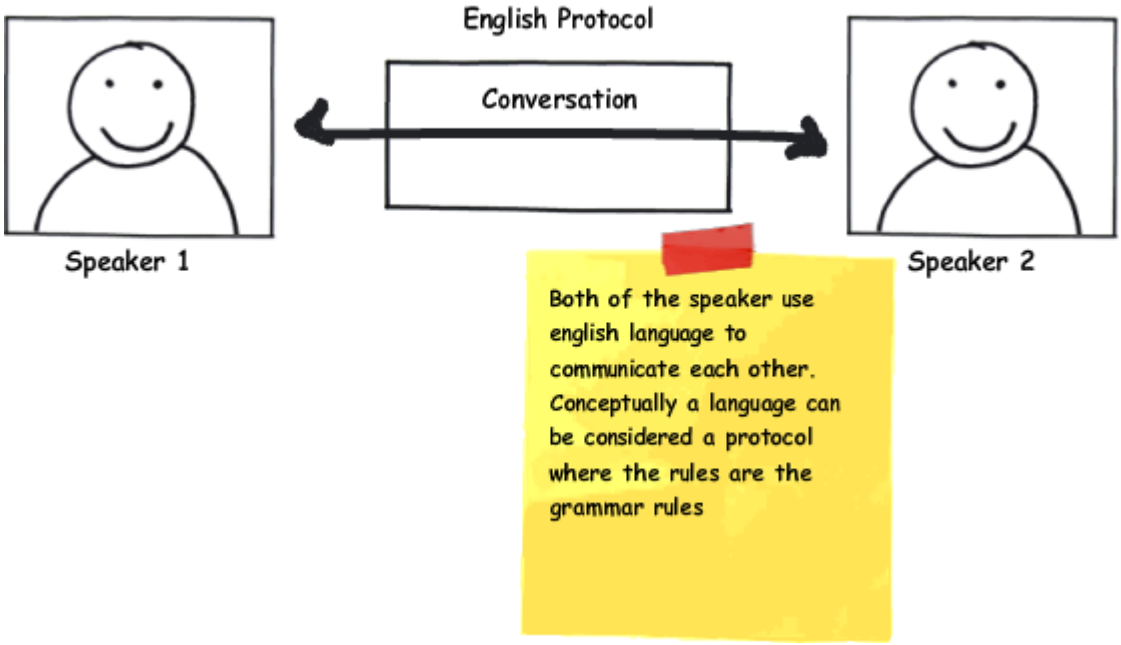
Let start from the real beginning.

A **Web Server** (like Internet Information Server/Apache/etc.) is a piece of software that enables a website to be viewed using **HTTP**. We can abstract this concept by saying that a web server is a piece of software that allows resources (web pages, images, etc.) to be requested over the HTTP protocol. I am sure many of you have thought that a web server is just a special super computer, but it's just the software that runs on it that makes the difference between a normal computer and a web server.

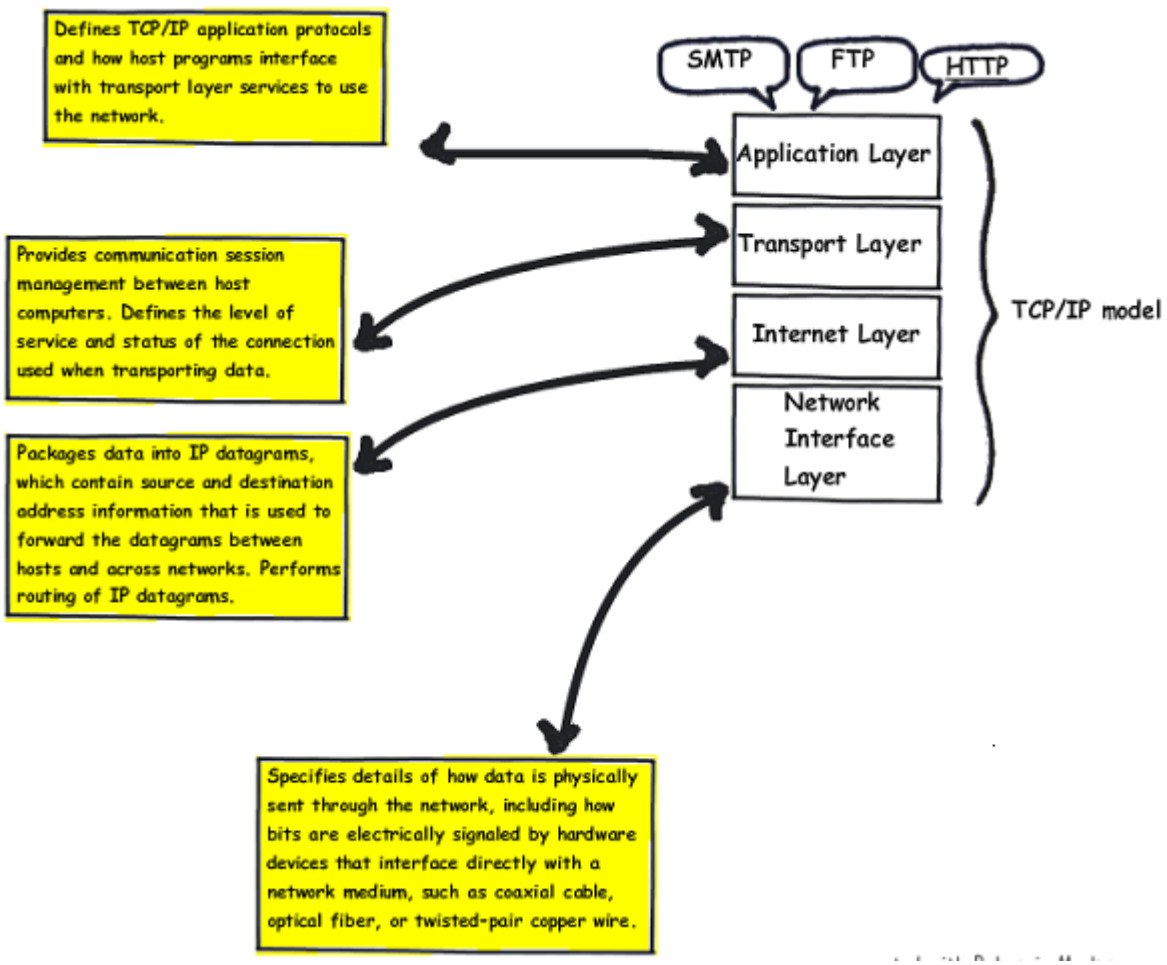


As everyone knows, in a **Web Communication**, there are two main actors: the **Client** and the **Server**.

The client and the server, of course, need a **connection** to be able to communicate with each other, and a common set of rules to be able to understand each other. The rules they need to communicate are called **protocols**. Conceptually, when we speak to someone, we are using a protocol. The protocols in human communication are rules about appearance, speaking, listening, and understanding. These rules, also called *protocols of conversation*, represent different layers of communication. They work together to help people communicate successfully. The need for protocols also applies to computing systems. A **communications protocol** is a formal description of digital message formats and the rules for exchanging those messages in or between computing systems and in telecommunications.



HTTP knows all the "grammar", but it doesn't know anything about how to send a message or open a connection. That's why HTTP is built on top of TCP/IP. Below, you can see the conceptual model of this protocol on top of the HTTP protocol:



TCP/IP is in charge of managing the connection and all the low level operations needed to deliver the message exchanged between the client and the server.

In this article, I won't explain how TCP/IP works, because I should write a whole article on it, but it's good to know it is the engine that allows the client and the server to have message exchanges.

HTTP is a **connectionless** protocol, but it doesn't mean that the client and the server don't need to establish a connection before they start to communicate with each other. But, it means that the client and the server don't need to have any prearrangements before they start to

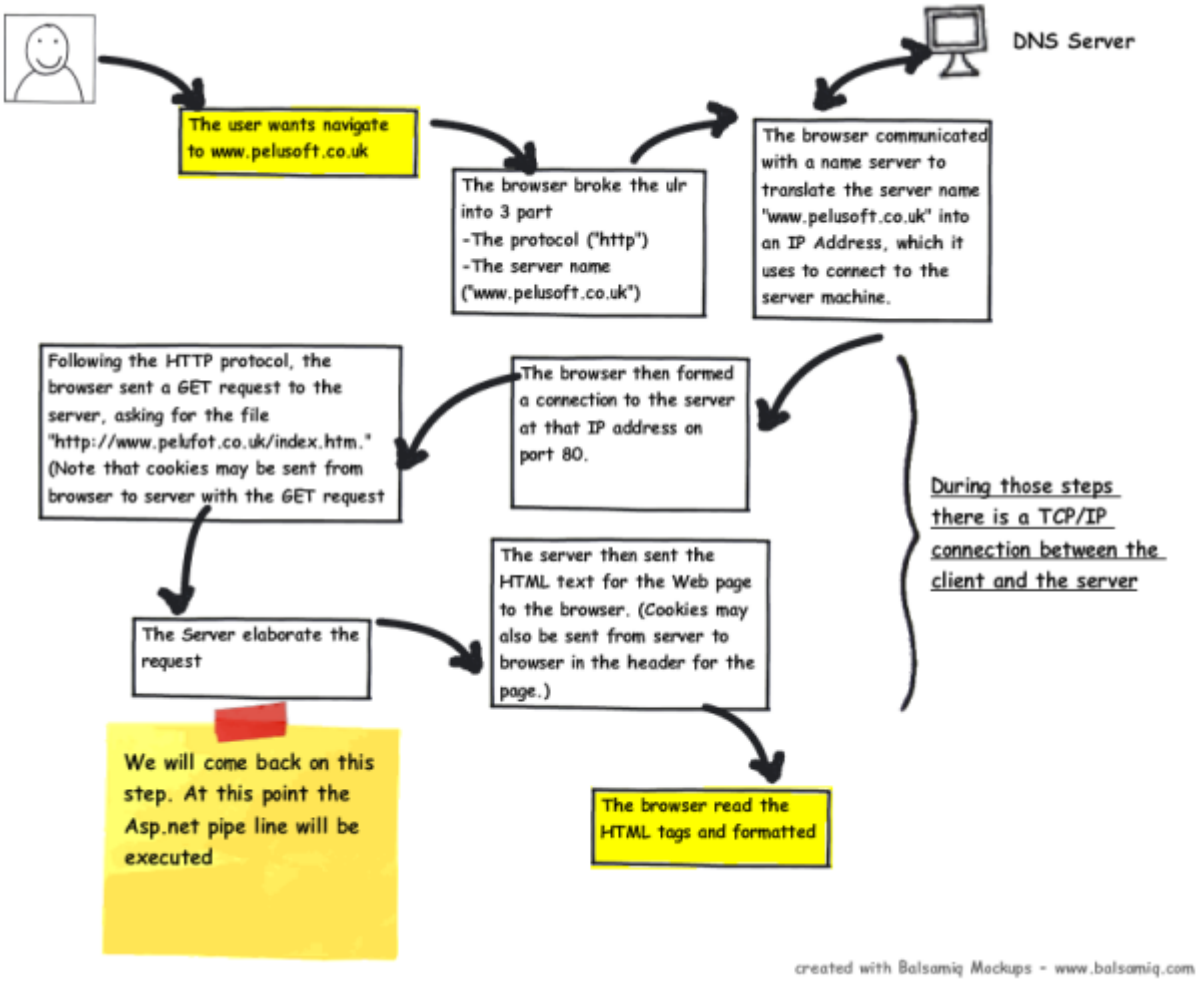
communicate.

Connectionless means the client doesn't care if the server is ready to accept a request, and on the other hand, the server doesn't care if the client is ready to get the response, but a connection is still needed.

In connection-oriented communication, the communicating peers must first establish a logical or physical data channel or connection in a dialog preceding the exchange of user data.

Now, let's see what happens when a user puts an address into the browser address bar.

- The browser breaks the URL into three parts:
    - The protocol ("HTTP")
    - The server name (www.Pelusoft.co.uk)
    - The file name (*index.html*)
- The browser communicates with a name server to translate the server name "www.Pelusoft.co.uk" into an IP address, which it uses to connect to the server machine.
- The browser then forms a connection to the server at that IP address on port 80. (We'll discuss ports later in this article.)
- Following the HTTP protocol, the browser sents a GET request to the server, asking for the file "*http://www.pelusoft.co.uk.com/index.htm*". (Note that cookies may be sent from the browser to the server with the GET request -- see *How Internet Cookies Work* for details.)
- The server then sents the HTML text for the web page to the browser. Cookies may also be sent from the server to the browser in the header for the page.)
- The browser reads the HTML tags and formats the page onto your screen.

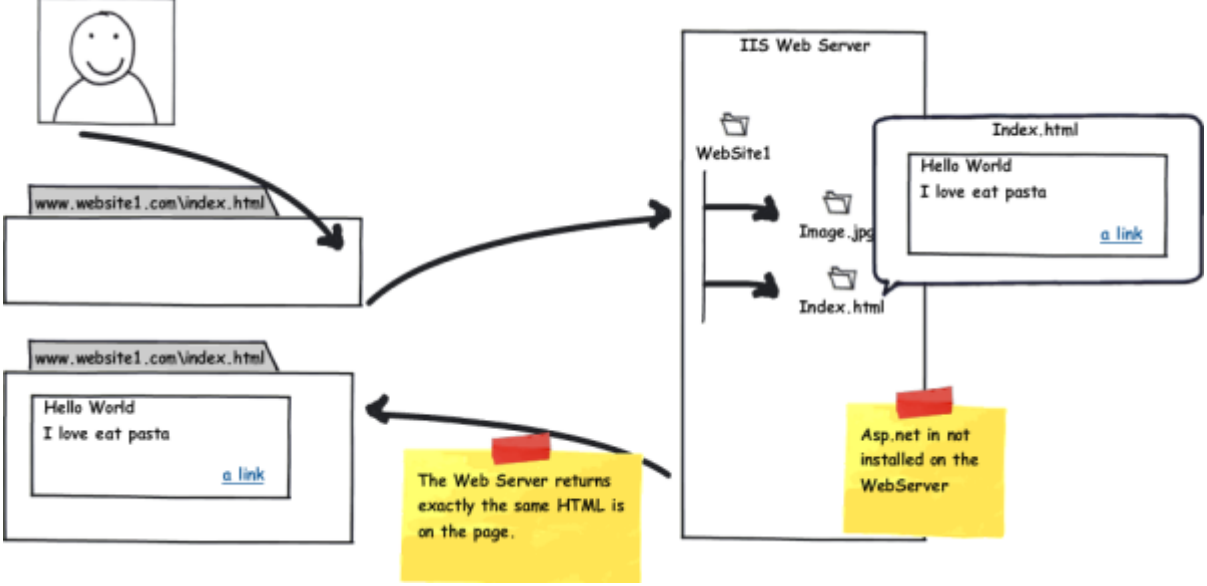


The current practice requires that the **connection** be established by the client prior to each request, and closed by the server after sending the response. Both clients and servers should be aware that either party may close the connection prematurely, due to user action, automated time-out, or program failure, and should handle such closing in a predictable fashion. In any case, the closing of the connection by either or both parties always terminates the current request, regardless of its status.

At this point, you should have an idea about how the **HTTP - TCP/IP** protocol works. Of course, there is a lot more to say, but the scope of this article is just a very high view of these protocols just to better understand all the steps that occur since the user starts to browse a web site.

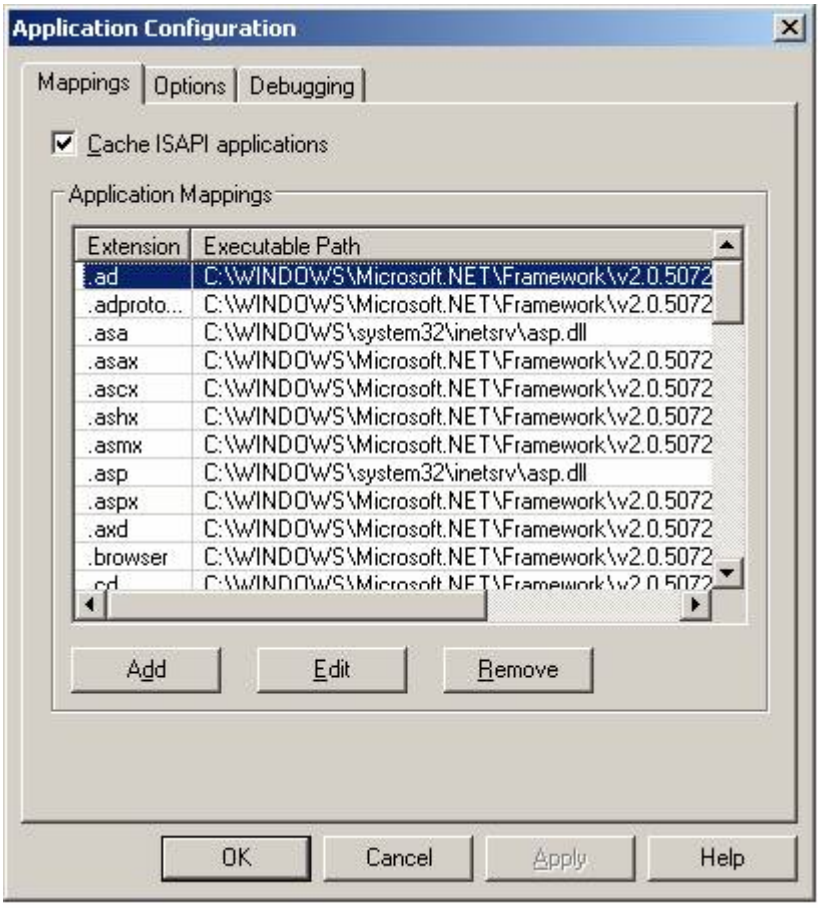
Now it's time to go ahead, moving the focus on to what happens when the web server receives the request and how it can get the request itself.

As I showed earlier, a web server is a "normal computer" that is running a special software that makes it a Web Server. Let's suppose that IIS runs on our web server. From a very high view, IIS is just a process which is listening on a particular port (usually 80). Listening means it is ready to accept a connections from clients on port 80. A very important thing to remember is: IIS is not ASP.NET. This means that IIS doesn't know anything about ASP.NET; it can work by itself. We can have a web server that is hosting just HTML pages or images or any other kind of web resource. The web server, as I explained earlier, has to just return the resource the browser is asking for.



ASP.NET and IIS

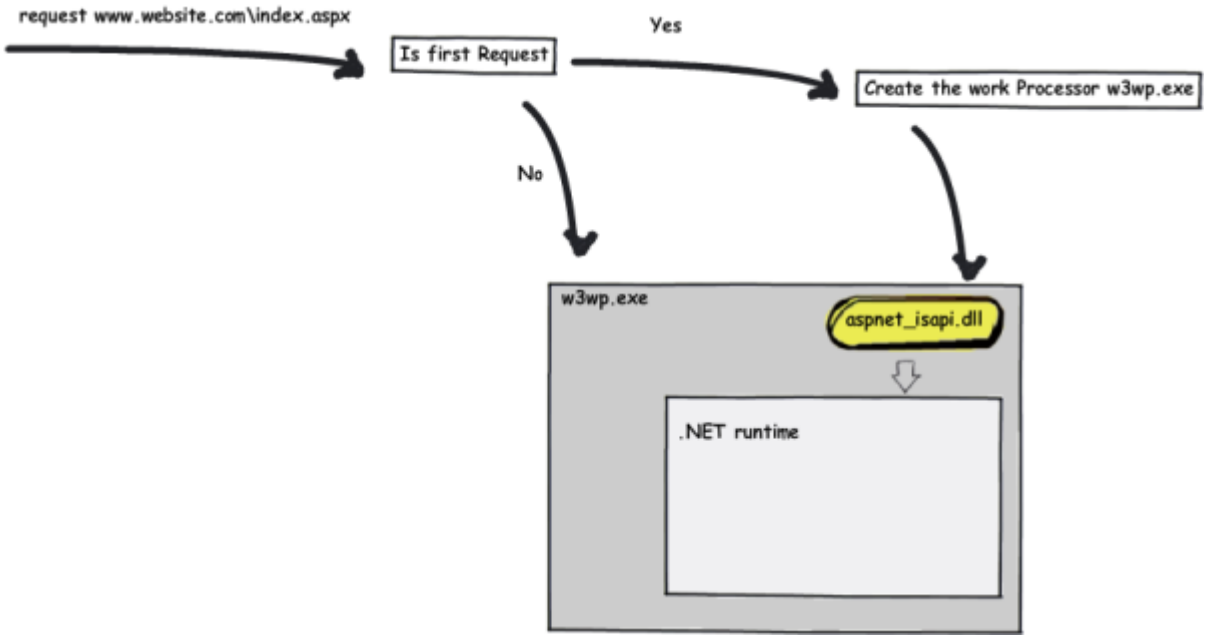
The web server can also support server scripting (as ASP.NET). What I show in this paragraph is what happens on the server running ASP.NET and how IIS can "talk" with the ASP.NET engine. When we install ASP.NET on a server, the installation updates the script map of an application to the corresponding ISAPI extensions to process the request given to IIS. For example, the "aspx" extension will be mapped to *aspnet_isapi.dll* and hence requests for an *aspx* page to IIS will be given to *aspnet_isapi* (the ASP.NET registration can also be done using *Aspnet_regiis*). The script map is shown below:



The ISAPI filter is a plug-in that can access an HTTP data stream before IIS gets to see it. Without the ISAPI filter, IIS can not redirect a request to the ASP.NET engine (in the case of a *.aspx* page). From a very high point of view, we can think of the ISAPI filter as a router for IIS requests: every time there is a resource requested whose file extension is present on the map table (the one shown earlier), it redirect the request to the right place. In the case of an *.aspx* page, it redirects the request to the .NET runtime that knows how to elaborate the request. Now, let's see how it works.

When a request comes in:

- IIS creates/runs the work processor (*w3wp.exe*) if it is not running.
- The *aspnet_isapi.dll* is hosted in the *w3wp.exe* process. IIS checks for the script map and routes the request to *aspnet_isapi.dll*.
- The request is passed to the .NET runtime that is hosted into *w3wp.exe* as well.



Finally, the request gets into the runtime

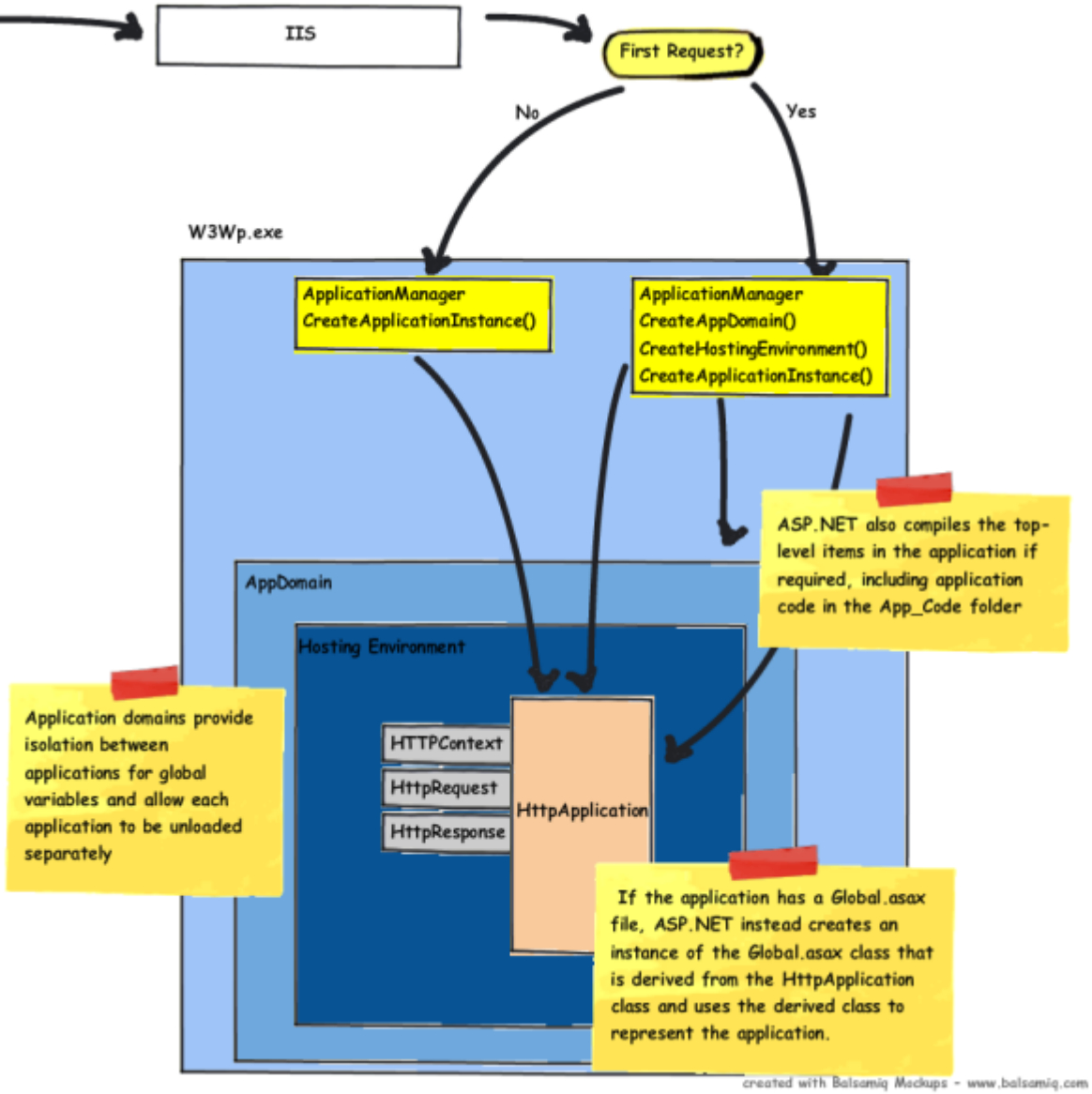
This paragraph focuses on how the runtime handles the request and shows all the objects involved in the process.

First of all, let's have a look at what happens when the request gets to the runtime.

- When ASP.NET receives the first request for any resource in an application, a class named `ApplicationManager` creates an application domain. (Application domains provide isolation between applications for global variables, and allow each application to be unloaded separately.)
- Within an application domain, an instance of the class named Hosting Environment is created, which provides access to information about the application such as the name of the folder where the application is stored.
- After the application domain has been created and the Hosting Environment object instantiated, ASP.NET creates and initializes core objects such as `HttpContext`, `HttpRequest`, and `HttpResponse`.
- After all core application objects have been initialized, the application is started by creating an instance of the `HttpApplication` class.
- If the application has a *Global.asax* file, ASP.NET instead creates an instance of the *Global.asax* class that is derived from the `HttpApplication` class and uses the derived class to represent the application.

Those are the first steps that happens against a client request. Most articles don't say anything about these steps. In this article, we will analyze in depth what happens at each step.

Below, you can see all the steps the request has to pass though before it is elaborated.



Application Manager

The first object we have to talk about is the **Application Manager**.

Application Manager is actually an object that sits on top of all running ASP.NET AppDomains, and can do things like shut them all down or check for idle status.

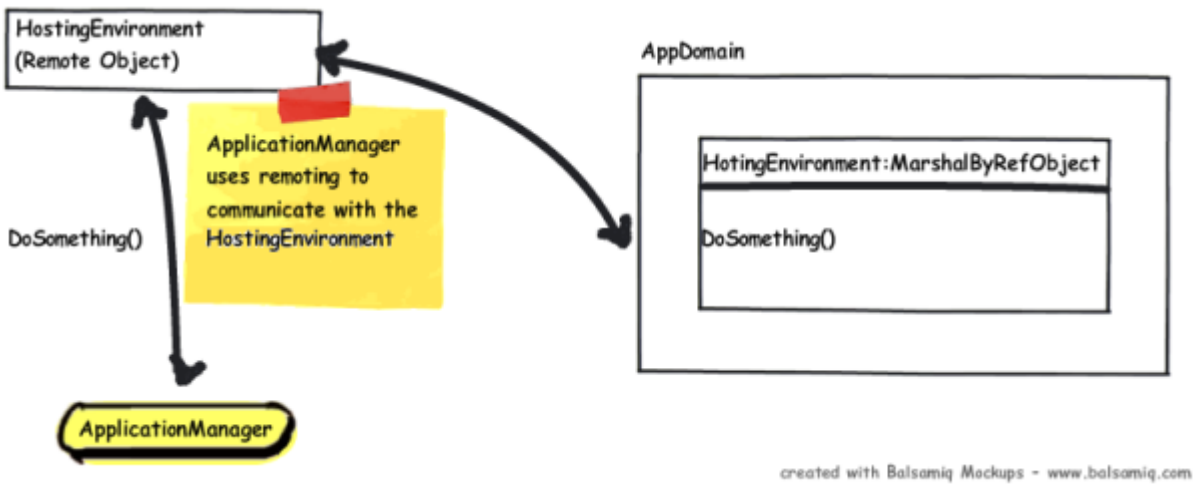
For example, when you change the configuration file of your web application, the Application Manager is in charge to restart the AppDomain to allow all the running application instances (your web site instance) to be created again for loading the new configuration file you may have changed.

Requests that are already in the pipeline processing will continue to run through the existing pipeline, while any new request coming in gets routed to the new AppDomain. To avoid the problem of "hung requests", ASP.NET forcefully shuts down the AppDomain after the request timeout period is up, even if requests are still pending.

Application Manager is the "manager", but the Hosting Environment contains all the "logic" to manage the application instances. It's like when you have a class that uses an interface: within the class methods, you just call the interface method. In this case, the methods are called within the Application Manager, but are executed in the Hosting Environment (let's suppose the Hosting Environment is the class that implements the interface).

At this point, you should have a question: how is it possible the Application Manager can communicate with the Hosting Environment since it lives in an AppDomain? (We said the AppDomain creates a kind of boundary around the application to isolate the application itself.) In fact, the Hosting Environment has to inherit from the `MarshalByRefObject` class to use Remoting to communicate with the Application Manager. The Application Manager creates a remote object (the Hosting Environment) and calls methods on it :-)

So we can say the Hosting Environment is the "remote interface" that is used by the Application Manager, but the code is "executed" within the Hosting Environment object.



HttpApplication

On the previous paragraph, I used the term "Application" a lot. `HttpApplication` is an instance of your web application. It's the object in charge to "elaborate" the request and return the response that has to be sent back to the client. An `HttpApplication` can elaborate only one request at a time. However, to maximize performance, `HttpApplication` instances might be reused for multiple requests, but it always executes one request at a time.

This simplifies application event handling because you do not need to lock non-static members in the application class when you access them. This also allows you to store request-specific data in non-static members of the application class. For example, you can define a property in the *Global.asax* file and assign it a request-specific value.

You can't manually create an instance of `HttpApplication`; it is the Application Manager that is in charge to do that. You can only configure what is the maximum number of `HttpApplication`s you want to be created by the Application Manager. There are a bunch of keys in the machine config that can affect the Application Manager behavior:

```
<processModel enable="true|false"
    timeout="hrs:mins:secs|Infinite"
    idleTimeout="hrs:mins:secs|Infinite"
    shutdownTimeout="hrs:mins:secs|Infinite"
    requestLimit="num|Infinite"
    requestQueueLimit="num|Infinite"
    restartQueueLimit="num|Infinite"
    memoryLimit="percent"
    webGarden="true|false"
    cpuMask="num"
    userName="<username>"
    password="<secure password>"
    logLevel="All|None|Errors"
    clientConnectedCheck="hrs:mins:secs|Infinite"
    comAuthenticationLevel="Default|None|Connect|Call|
    Pkt|PktIntegrity|PktPrivacy"
    comImpersonationLevel="Default|Anonymous|Identify|
    Impersonate|Delegate"
    responseDeadlockInterval="hrs:mins:secs|Infinite"
    responseRestartDeadlockInterval="hrs:mins:secs|Infinite"
    autoConfig="true|false"
    maxWorkerThreads="num"
    maxIoThreads="num"
    minWorkerThreads="num"
    minIoThreads="num"
    serverErrorMessageFile=""
    pingFrequency="Infinite"
    pingTimeout="Infinite"
    maxAppDomains="2000"
/>
```

With `maxWorkerThreads` and `minWorkerThreads`, you set up the minimum and maximum number of `HttpApplication`s.

For more information, have a look at: ProcessModel Element.

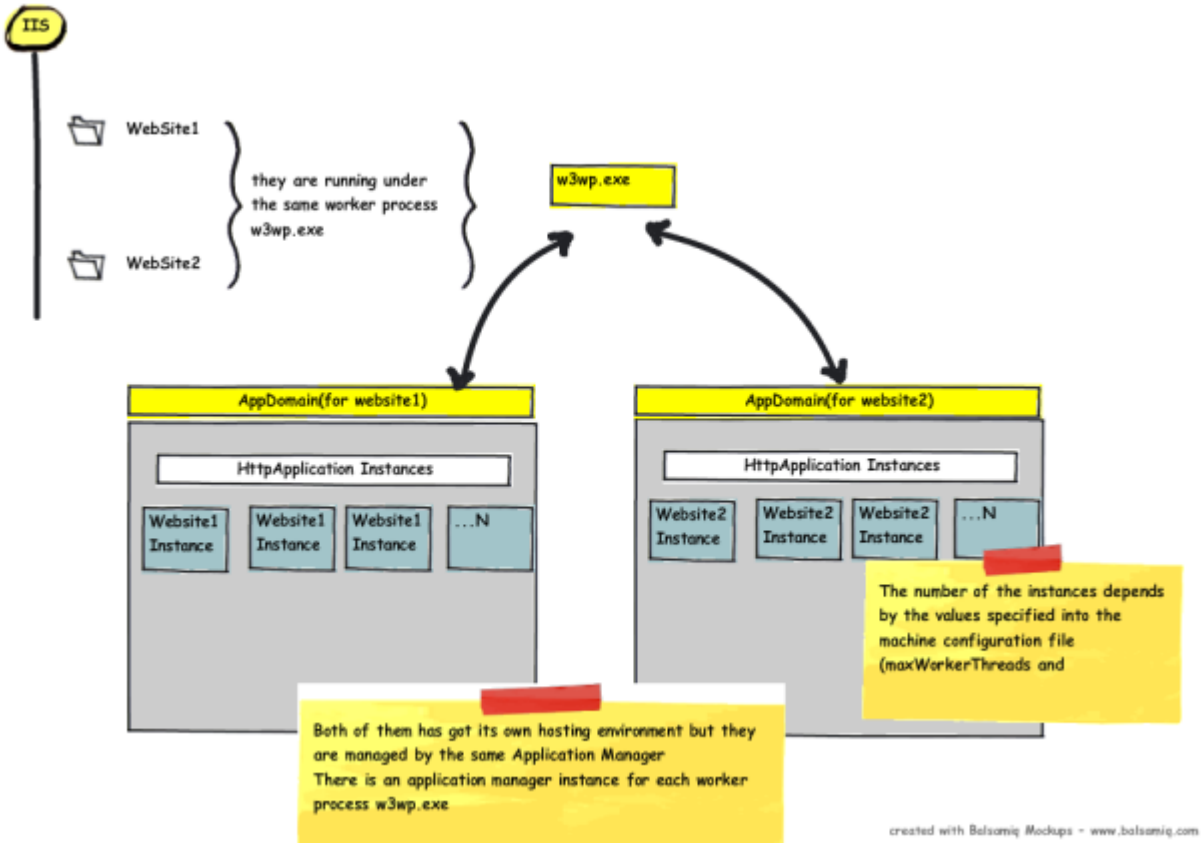
Just to clarify what we have said until now, we can say that against a request to a WebApplication, we have:

- A Worker Process *w3wp.exe* is started (if it is not running).
- An instance of `ApplicationManager` is created.
- An ApplicationPool is created.
- An instance of a Hosting Environment is created.

- A pool of `HttpAplication` instances is created (defined with the *machine.config*).

Until now, we talked about just one WebApplication, let's say WebSite1, under IIS. What happens if we create another application under IIS for WebSite2?

- We will have the same process explained above.
- WebSite2 will be executed within the existing Worker Process *w3wp.exe* (where WebSite1 is running).
- The same Application Manager instance will manage WebSite2 as well. There is always an instance per Work Proces *w3wp.exe*.
- WebSite2 will have its own AppDomain and Hosting Environment.
- Within a new AppDomain will be run instances of WebSite2 (`HttpApplication` instances).



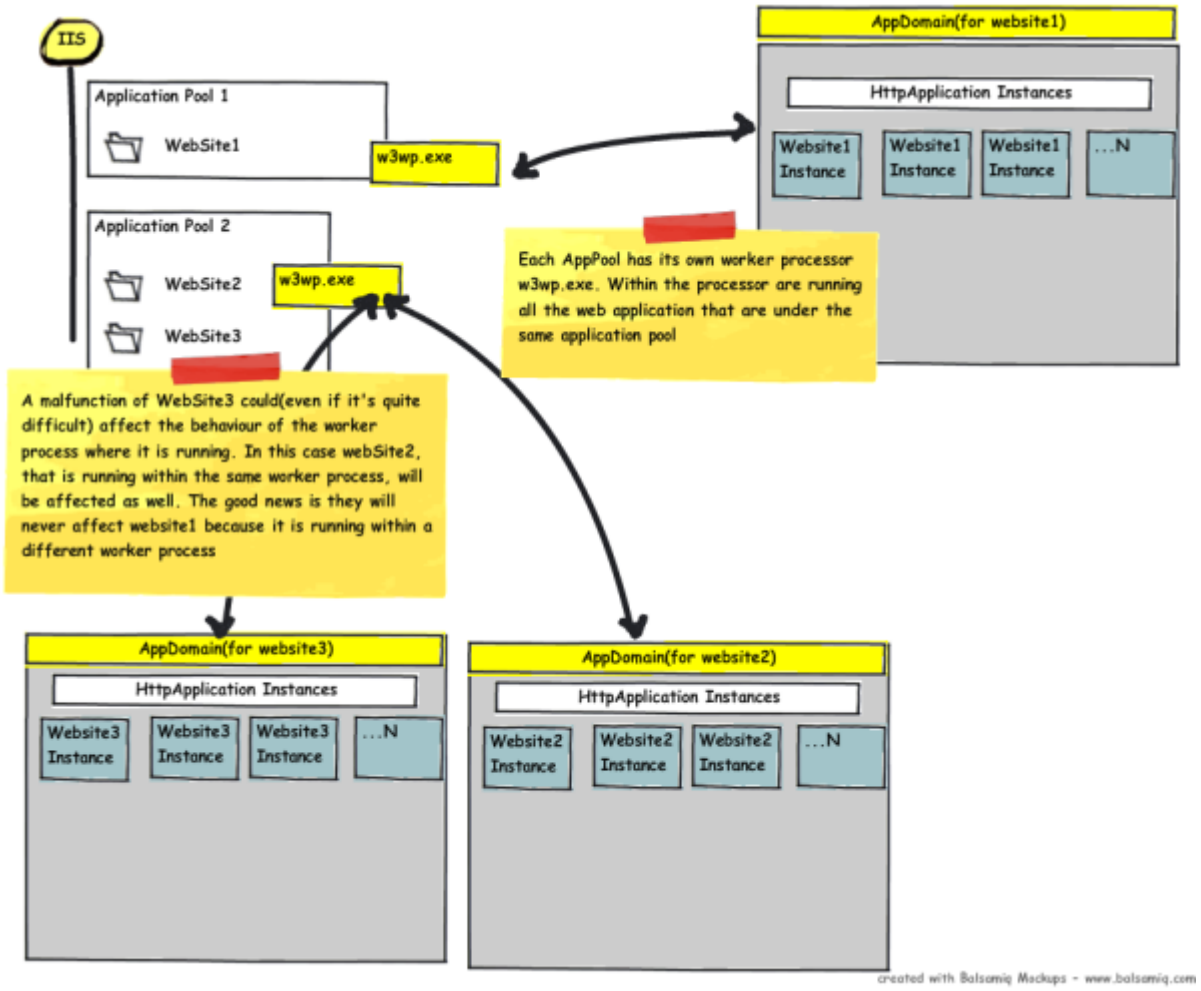
It's very important to notice that each web application runs in a separate AppDomain so that if one fails or does something wrong, it won't affect the other web apps that can carry on their work. At this point, we should have another question:

What would happen if a Web Application, let's say WebSite1, does something wrong affecting the Worker Process (even if it's quite difficult)?

What if I want to recycle the application domain?

To summarize what we have said, an AppPool consists of one or more processes. Each web application that you are running consists of (usually, IIRC) an Application Domain. The issue is when you assign multiple web applications to the same AppPool, while they are separated by the Application Domain boundary, they are still in the same process (*w3wp.exe*). This can be less reliable/secure than using a separate AppPool for each web application. On the other hand, it can improve performance by reducing the overhead of multiple processes.

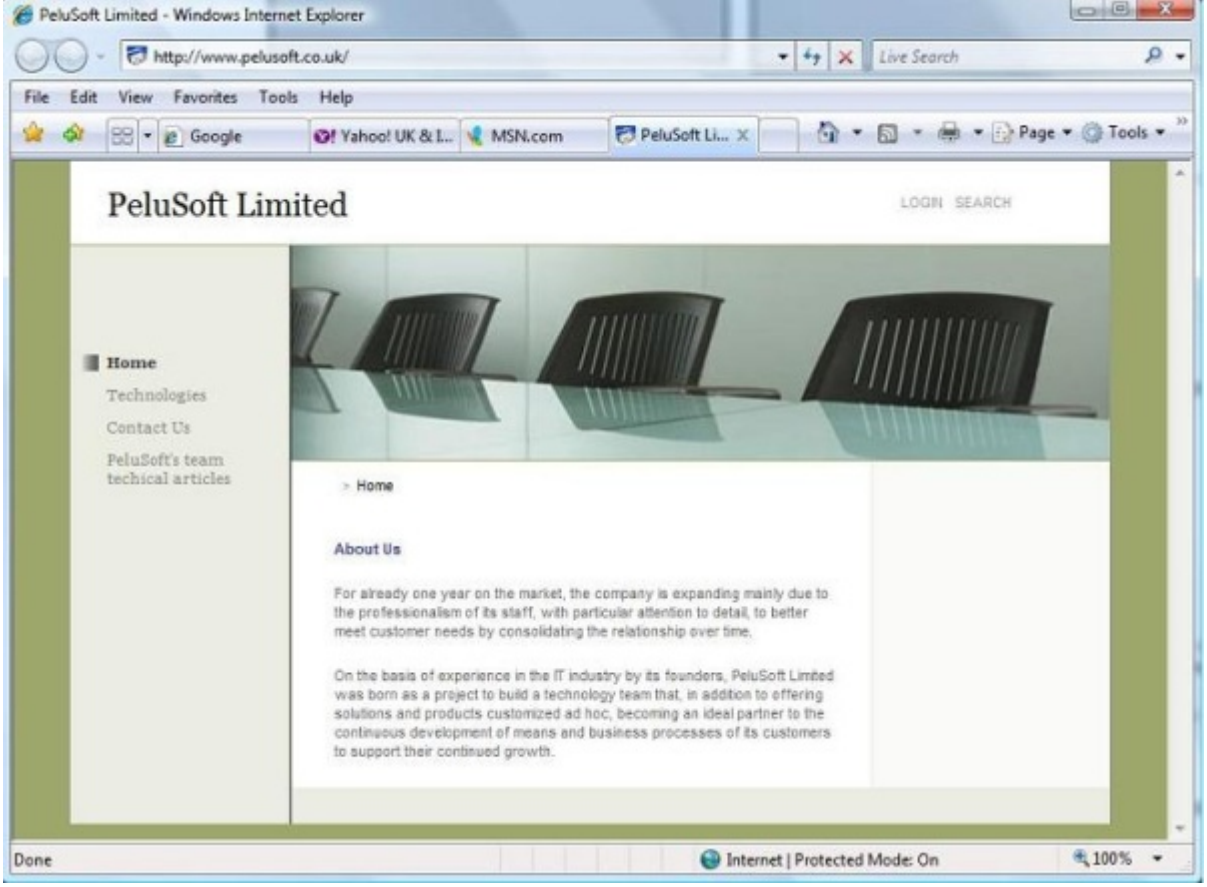
An Internet Information Services (IIS) application pool is a grouping of URLs that is routed to one or more worker processes. Because application pools define a set of web applications that share one or more worker processes, they provide a convenient way to administer a set of web sites and applications and their corresponding worker processes. Process boundaries separate each worker process; therefore, a web site or application in an application pool will not be affected by application problems in other application pools. Application pools significantly increase both the reliability and manageability of a web infrastructure.



I want to better understand the difference between an AppDomain and an Application Pool

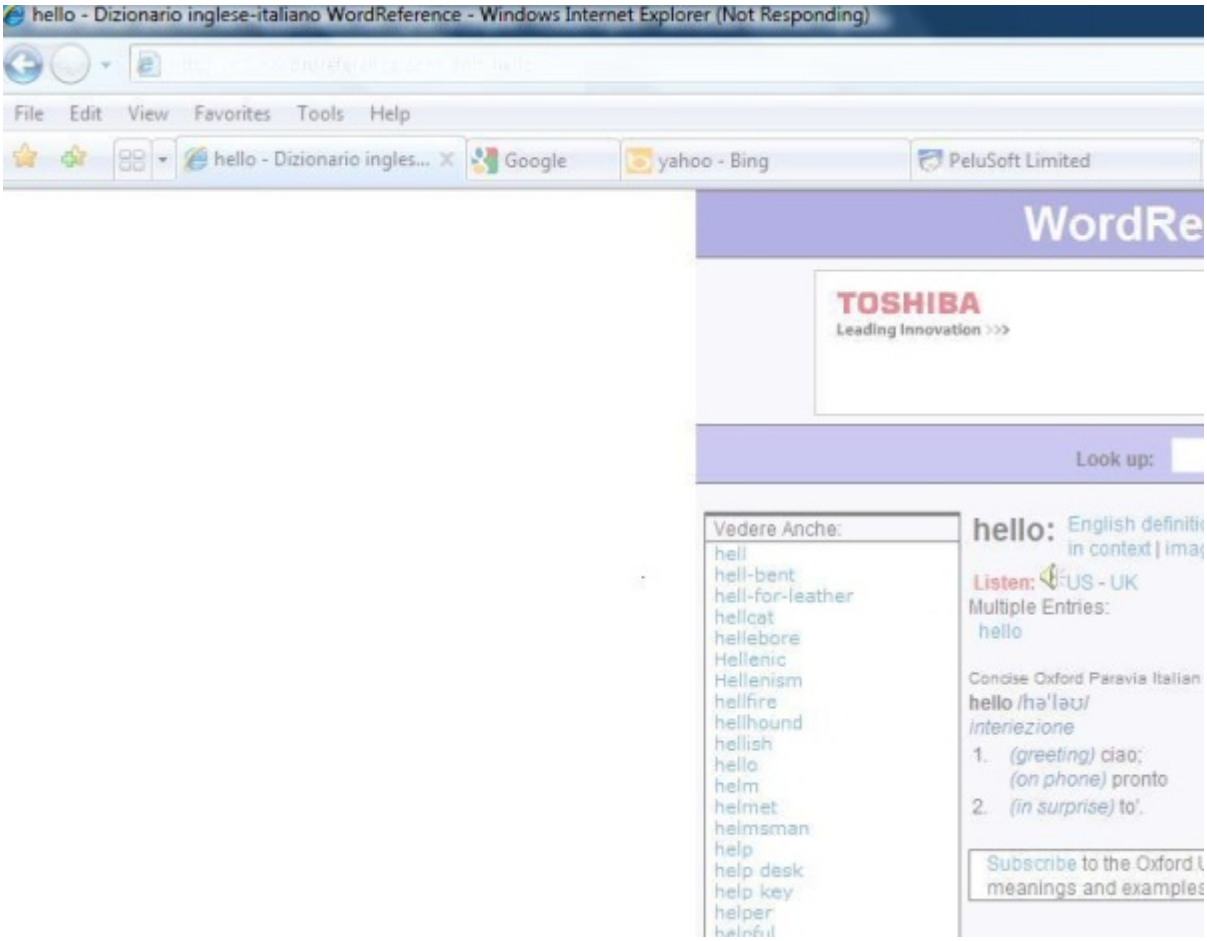
Just for better understanding, let's say you are working with Internet Explorer. As you know, you can open more than one tab for browsing more than one web site in the same instance of IE.

Let' say you open four tabs browsing different web sites, so that each tab will show a different web site.

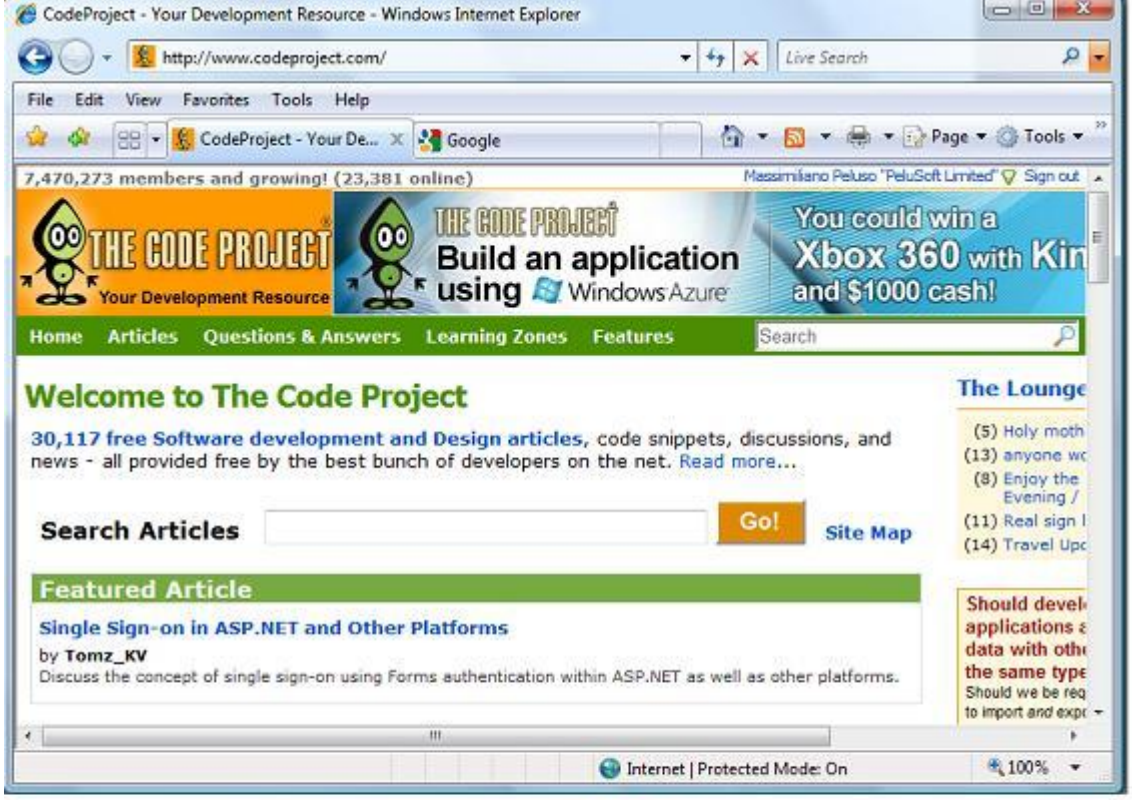


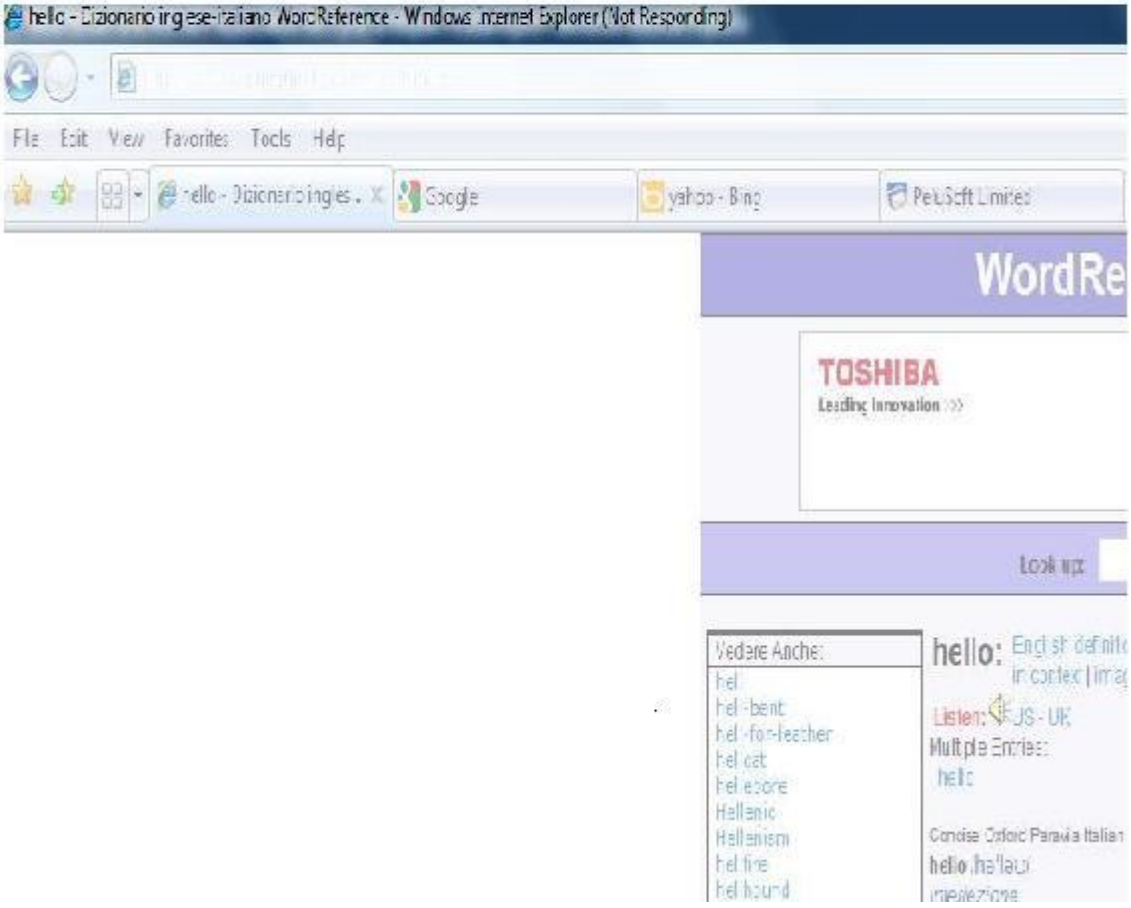
After lots of pints of beer, let's assume that each tab is running within a different App Domain and each web site we are browsing is a different web app under IIS. All the web sites we are browsing are sharing the same instance of IE (*iexplorer.exe*). In this example, let's compare the IE instance (*iexplorer.exe*) to the ASP.NET Worker Process (*w3wp.exe*). If one tab is stuck for any reason (as sometime happens :-)), it's going to affect all the other tabs running under the same *iexplorer.exe* instance (some times, IE gets stuck and it doesn't respond to any interaction anymore, showing *Not responding*), so you are forced to kill the process to carry on working.

By killing the process, you will not able to browse any of the web sites you were browsing, even if it was just one process that made got IE stuck.



So what can we do if we want to isolate web site browsing at process level so that if one web site does something wrong, we can carry on browsing the other web sites? Simple, you can open two instance of IE and open two tabs on each instance. In this case, two *iexplore.exe* instances will be running, and they will be completely independent of each other: if a tab is stuck on one instance of Internet Explorer, you can kill that process, but the other one will carry on to work.





At this point, you should have an idea of what is happening in depth during a web request.

Of course, I should say lots more regarding this topic, but the scope of this article is just to give you an idea of what is really happening internally in IIS/ASP.NET and all the objects that are created to satisfy a web request.

References

- ASP.NET Application Life Cycle Overview for IIS 5.0 and 6.0
- What ASP.NET Programmers Should Know About Application Domains
- A Low-Level Look at the ASP.NET Architecture

License

Share

About the Author

Massimiliano Peluso "WeDev Limited"

Architect PeluSoft Limited
United Kingdom

I have been fascinated by software development since I was 10 years old. I'm proud of learned the first programming language with an Olivetti PC 128S based on Microsoft BASIC. I'm a Software Architect/Senior Developer with a strong background of all the developing Microsoft's technologies.

You may also be interested in...

Keeping Up With PHP

The Business Case for Earlier Software Defect Detection and Compliance

ASP.NET Life Cycle Overview

Using the Intel® Edison Module to Control Robots

ASP.NET Application and Page Life Cycle

SAPrefs - Netscape-like Preferences Dialog

Comments and Discussions

95 messages have been posted for this article Visit **https://www.codeproject.com/Articles/121096/Web-Server-and-ASP-NET-Application-Life-Cycle-in-D** to post and view comments on this article, or click **here** to get a print view with messages.